Measuring Interaction Design before Building the System: a Model-Based Approach

Giorgio Brajnik[1,2]
[1]Dip. di Matematica, Informatica e Fisica
Università di Udine, Italy
brajnik@uniud.it

Simon Harper[2]
[2]Computer Science School
University of Manchester, UK
simon.harper@manchester.ac.uk

ABSTRACT

Early prototyping of user interfaces is an established good practice in interactive system development. However, prototypes cover only some usage scenarios, and questions dealing with number of required steps, possible interaction paths or impact of possible user errors can be answered only for the specific scenarios and only after tedious manual inspection.

We present a tool (MIGTool) that transforms models of the behavior of a user interface into a graph, upon which usage scenarios can be easily specified, and used by MIGTool to compute possible interaction paths. Metrics based on possible paths, with or without user navigation errors, can then be computed. For example, when analyzing four mail applications, we show that Gmail has 3 times more shortest routes, has twice more routes that include a single user error, has routes with 13% fewer steps, but has also optimal routes with the smallest probability to be chosen.

Without MIGTool, this kind of analysis could only be done after building some prototype of the system, and then only for specific scenarios by manually tracing user actions and relative changes to the screens. With MIGTool the exploration of suitability of a design with respect to different scenarios, or comparison of different design alternatives against a single scenario, can be done with just a partial specification of the user interface behavior.

This is made possible by the ability to associate scenarios steps to required user actions as defined in the model, by an efficient strategy to identify complete execution traces that users can follow, and by computing a range of diverse metrics on these results.

Keywords

User Interfaces; Evaluation; Statecharts: UML; Metrics; Usability.

Categories and Subject Descriptors

H.5.1 [**Information interfaces and presentation (e.g., HCI)**]: Multimedia Information Systems.; H.5.2 [**Information interfaces and presentation (e.g., HCI)**]: User Interfaces.

EICS'16, June 21 - 24, 2016, Brussels, Belgium

ISBN 978-1-4503-4322-0/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2933242.2933246

1. INTRODUCTION

We present an approach that allows a designer to assess interaction design qualities such as efficiency, error-proneness and recovery from errors. Key importance is given to the ability to (a) understand how supportive a user interface (UI) is with respect to user efficiency; (b) understand how prone the UI is to user navigation errors; (c) understand how recoverable the UI is from those errors; and, (d) perform objective quantitative comparisons of different designs to support construction and engineering of interactive systems. All this can be done before developing prototypes of the UI, similarly to what is claimed and done in [35]: "... the analysis of solutions in the early phases of development is crucial as their results can then be iteratively injected in the development cycle".

To explain our work we present a comparison of four web mail front ends. We have chosen this domain because it is easy to understand and yet several questions cannot be easily answered. We applied the same techniques also in other domains, such as HVAC (Heat, Ventilation and Air Conditioning) and other embedded UIs.

Developing good UIs for web or mobile applications is a complex and expensive endeavor. One reason is the combination of devices, interaction modalities and workflows that need to be supported. Adoption of usage-centered development practices and following established design principles [11] is a sound way to tackle the problem. In particular very effective techniques are prototyping, to explore part of the five-dimensional fidelity space [23, 8], paired with usability investigations, based on user testing or heuristic evaluations [32, 34].

However, prototypes are usually developed with certain tasks in mind, and therefore are quite restricted in terms of depth, breadth, dynamics and data. Furthermore, in addition to the possible bias introduced by prototypes, usability results are always surrounded by a cloud of uncertainty, due to subjectivity introduced by participants and facilitators or by other contingency factors involved in the analysis. Thus, although a significant effort needs to be directed to develop and use prototypes, less than optimal results are obtained.

Even worse, several questions cannot be easily answered. Given one or more potential designs and some usage scenarios, interesting questions include: "How many different routes can be followed by the user to carry out the scenario?", "Which are the shortest ones?", "If a user makes a mistake, would he or she be able to recover?", "How many steps would the recovery require?". When designing and evaluating embedded UIs (such as when dealing with plane's cockpits [3]), other relevant questions include "How would the above properties change if we add a certain a widget?", or "... if we replace a widget with another?". At the moment, these straightforward questions are quite complicated to answer. In fact, they

require developing prototypes, inspecting them, manually tracking which screens and widgets are used at which stage, followed by a systematic manual analysis.

This should not be the case, however, because answers to these questions could be automatically computed. Such a system could provide important insights to a designer, and support assessments of potential user flexibility, user efficiency, error proneness, ability to recover, compactness and consistency of a design.

Our approach is based on UML state machine models of UIs which are automatically processed to produce *interaction graphs*. These graphs are then used to specify interaction scenarios and to unfold the possible interaction paths (called *execution traces*) that are compatible with the specific scenarios being considered. Traces are processed to produce a dashboard with different results. Except for development of models and specification of the desired scenarios, which have to be done manually, the other steps are totally automatic; models of a design (such as the ones shown below) can be developed in a matter of a couple of hours.

Our contribution consists of (a) the idea of using a UML state machine model to support specification of interaction scenarios, (b) the development of a tool (MIGtool, Measuring Interaction Graphs Tool) that, based on models, transforms scenario specifications into interaction paths, and (c) the definition of metrics that provide concise, precise and objective measures of the interaction structure entailed by a user interface. The examples presented below show that among four web mail applications, and with respect to a typical usage scenario, Gmail is the most efficient and flexible UI, with the best ability to recover from user errors, but only for users that possess a certain level of proficiency. In fact, Gmail has the largest number of shortest paths (when users are supposed to make 0 or 1 error); when users make 2 or more errors the number of paths drops significantly, which indicates that the error-proneness of the UI has reduced; in the best case, Gmail features also the shortest paths, requiring 13% fewer steps than other applications; however, the probability that a user hits an optimal path with Gmail is 10 times smaller than the best of the other applications, and the probability that a random walker hits a state that is not due to an error is 10% smaller than the best of the other applications. These values suggest that Gmail offers more ways to accomplish tasks included in the scenario, that comparably more of these ways do not involve extra steps, that they require fewer steps, and that it might be more difficult for novice users to exploit the most efficient ways. Further inspections show that some differences are due to the slightly different interaction structures adopted for uploading attachments. Thus, redesigning the methods with which competing applications deal with attachments could improve their usability. If Gmail didn't exist yet, by using MIGtool its designers could obtain these answers well before developing prototypes and performing usability studies.

Other examples discussed below show what is gained when a new feature is added to an embedded automotive system.

2. BACKGROUND

The literature on using state-transition networks for specifying or analyzing the behavior of UIs is vast. We conducted a systematic-style literature review, using Google Scholar and queries with combinations of these phrases: "user interface", "path analysis", "user trace", "interaction trace", "navigation", "markov chain", "markov model", "state transition", "statechart", "metric", "measure". For each query we analyzed title and abstract of the first 50 hits and appraised their relevance based on whether the paper discusses approaches for measuring user interfaces in the context of usability and whether it relies on a state transition model. This resulted in 78 full-text papers that were later on re-analyzed against the same criterion, leading to several of papers mentioned below.

Usage of state-transition networks to model UI behavior in order to draw usability conclusions dates back at least to [30]. In it, Parnas claimed that several kinds of usability problems could be avoided if the designer adopted a design framework where states and their transitions are made explicit.

In many cases *statecharts* are used, a generalization of finite state automata. Horrocks showed how statecharts can be used to model and specify the dynamics of typical desktop UIs [18]. While providing many interesting insights on how and why one should use statecharts to do so, this nice work does not address how such a specification could be *automatically* processed. This idea was later on expanded by Thimbleby [38]; statecharts are seen as representations that allow a designer to fully appreciate how devices behave. The overall stance is that "If you don't understand the logic conveyed by a statechart model of a user interface, then you don't understand the behavior of that user interface".

WebML [10] is one of the most successful model-driven approaches to web development (UIs and backend systems), with industrial traction and a large number of publications. The language is based on state transitions and is targeted to automatic generation of data-intensive web applications. Many other similar approaches involve or are based on task or activity models [25, 31, 20, 14, 24]. A recent OMG standard, called Interaction Flow Modeling Language (IFML) [26], derives from WebML and focuses specifically on user interaction. IFML is a language for specifying the structure of a user interface and its behavior. It offers most of the abstractions that are available in statecharts, mixed with the ability to specify so-called "components" that are used to display and manipulate data (to display details of a item, to display or select lists of items, to input an item). None of these approaches, however, focus directly on measures of usability.

A different route for the problem of generating UIs is followed in [15]. Authors assume that the UI to be generated is used to supervise and to monitor an underlying machine (*e.g.*, autopilot of a plane) which is modeled as a statechart. After assuming that the behavior of the UI can also be modeled as a statechart, they devised an algorithm that checks whether the two models are compatible, and that refines the UI model so that its states and transitions are minimized while still allowing a correct manipulation of the underlying machine. Application of such a technique leads to UIs that are correct by-construction.

Finite state representations have been used also as a conceptual framework for writing the code of widgets so that events and event handlers in the UI can more easily be conceived, developed and verified (*e.g.*, [2]). More sophisticated model-based approaches employ statecharts to orchestrate the behavior of different components of a user interface [41]. These models, however, are not used to support usability analysis of any kind.

On the other hand the literature on metrics is also large. Lostness [36, 28] is a metric for measuring the degree to which users become lost in the information space, and it considers the notion of *deviation*. Defined specifically for hypertext systems, lostness is a user performance measure which is a function of the number of visited nodes, the number of different nodes that were visited, and the number of required nodes. This measure of efficiency is usually applied to traces of actual users, and is argued to be suitable for hypertext systems because the predominant task is browsing information, rather than trying to achieve specific goals. It is suitable therefore when some of the following three assumptions can be relaxed: that there is a task to complete, that there is a correct way to carry it out, that the purpose of the system is to support users in carrying out their tasks.

The notion of *deviation from optimal solution* is discussed only in the context of tools for route planning in 3D navigation. In this paper we provide our own definition of deviation, which applies to interaction with the UI; we provide also our notion of *potential execution traces* across states of the UI, and the length of these traces can be used as a measure of efficiency.

In [12], hierarchical task models (specified with HAMSTERS) are used to specify how pilots are expected to interact with airplane cockpit interfaces in tasks such as changing value to some flight parameter. The paper in particular discusses how these task models can be used to specify alternative ways to achieve the same goal, and analyses the consequences on safety and reliability of the resulting system. Compared to MIGTool, this approach cannot take advantage of a model of the system and instead relies only on models of the tasks. The consequence is that analyzing alternative ways to achieve the same goal requires developing different models; in addition, exploring suitability of a given system with respect to certain tasks is complex because no model of the system is available. Finally, using *only* hierarchical tasks as a modeling language limits the richness of the scenarios being analyzed, because it is difficult to take advantage of contexts and rich control flows that are available with state-transition representations, especially with UML state diagrams.

A review of usability measuring practices [17] lists, under the headings *measures of efficiency/usage patterns*, number of keystrokes, mouse clicks, and visited objects as possible metrics. Of course also the time needed to carry out individual interaction steps counts and it depends on many aspects, such as the time to decide what goal to achieve, to understand how to carry it out, to locate the actions to do it, to perform the action, to attend to the feedback. Layout of the user interface, Fitt's law considerations, cognitive state of the user, physical and environmental aspects affect such times. For these reasons we opted, in MIGtool, to only focus on required steps to complete a task, not on completion time.

A usability analysis method capable to analytically predict task completion times from a storyboard of the UI is based on using CogTool [5]. CogTool relies on the ACT-R cognitive modeling engine, and allows a designer to setup a low-fidelity prototype of the UI. After deciding which interaction modality and which widgets are used to implement actions, the designer gets an estimation of how long a skilled user would spend on each step. CogTool takes care of adding extra "mental" steps before certain patterns of provided steps, according to the cognitive theory underlying ACT-R. As a result, users of CogTool obtain the breakdown of the times required by the task. Our method is less precise: it does not provide expected completion times. However, with our method, a designer can analyze a large part of the UI, get information about possible problems in some areas, and only then devolve more resources in building storyboards and in making assumptions regarding widgets so that specific execution paths previously identified can be analyzed with CogTool. In a sense, the output obtained with our method could be used to make informed decisions as to what to analyze next with CogTool. Other approaches, like [37], start from the assumption that the user interface exists already, and that it can *ripped* to reverse engineer a behavior model. These assumptions are not needed when using MIGtool.

Another approach for analyzing user actions that has a strong cognitive background is SNIF-ACT [13]. However it has been conceived and applied to information websites, where the notion of *information scent* bears upon interaction. We opted for not applying such concepts to the interaction with a web application, where most of the actions are not aimed at information finding.

Markov models, *i.e.* directed graphs where edges leaving a vertex are associated to a probability distribution, were used in [39] as a means to perform usability analysis as early as possible, even before building prototypes of the UI. Vertices represent states of the UI and edges correspond to user actions (such as pressing a button). Probabilities can be used as a model of user knowledge: equiprobable actions correspond to a knowledge-free user, whereas when some actions have a very low probability it means that for that user the action is unlikely to be executed. Simple mathematical operations on the transition matrix of the model give the probability that after n steps from a given initial state the UI is in a given state. With Markov models, by manipulating probabilities, the designer can plot the number of required steps as a function of how close the probabilities are to the designer's "perfect" knowledge. Examples discussed in the paper cover several devices, ranging from a simple torch (with 4 states), a microwave cooker (6 states), a mobile phone (152 states). Notice that those are all push-down devices with a fixed set of buttons. This is obviously not the case for UIs of information systems, where buttons may change screen by screen and there could also be an arbitrary number of links. This makes it more difficult to specify the transition matrix. Our approach is based on statecharts, a language that in practice is more powerful than Markov models, making it easier to specify the UI behavior, especially in cases where the set of buttons change over time. While our examples do not make use of probabilities, this is very easy to cope with (see the Discussion at the end of the paper for some of the benefits that doing this could bring). Similarly to [39], our approach could be used when conservative results that do not rely on psychological assumptions are sought.

A discussion of social network analysis metrics applied to interaction design is provided in [40]. Once more, a UI is modeled in terms of directed graphs (vertices are states and edges are actions), and various centrality metrics are used to draw conclusions that bear upon usability. For example, centrality measures (such as Sabidussi, eccentricity, betweenness) can be used to identify states that are good places to start from to get to other states. Other metrics, such as edge betweenness, can be used to identify actions that are important because most of the shortest paths go through these actions. The paper presents compelling examples of using this technique to identify shortcomings in the design of infusion pumps. In our work we automatically generate graphs from statechart models, and on some of them we computed these metrics. In case studies presented below we were not able to draw sensible conclusions from the values we obtained (for example from the models presented below). One possible explanation rests on the different types of models: in our case they reflect the variety and flexibility with which "buttons" can be used in modern web applications. For infusion pumps the UI is more constrained in how a task can be carried on, and this difference might reflect on the usefulness of those metrics.

In [16] several approaches to analyze streams of user events are discussed and compared. It is interesting to realize that this is, in a sense, the inverse problem of the one we tackle in this paper: we want to compute a subset of the *possible* streams of user events given a specification of the UI, rather than trying to abstract general properties from *actual* streams of events.

3. GENERATION OF TRACES

Traces are potential paths (*i.e.*, sequences of connected states of the model) that users can follow when performing a given scenario. The generation process encompasses the following steps: (1) processing the model and automatic flattening of the statechart model; (2) manual definition of the interaction scenario; (3) automatic generation of execution traces; and (4) interactive analysis of results.

3.1 Processing models

MIGtool takes as input UML statecharts, which are a generalization of finite state automata (FSA) specified with an expressive language that includes hierarchical levels of abstraction, concurrent regions, states and pseudostates, guards, and an extended state notion based on an arbitrary underlying computational model.

By using statecharts, the behavior of UIs can be represented by associating states to screens and particular configurations of widgets, and transitions to actions performed by users or by the system itself [18]. In the classification reported in [16], actions belong to the "abstract interaction events" category.

Because statechart models take advantage of abstraction features and a rich set of connecting pseudostates, they are not suitable to be directly processed to compute metrics. For this reason, MIGtool first *flattens* the model. Flattening is a process often used when statecharts have to be automatically processed [7], and it means to produce a FSA that is behaviorally equivalent to the original statechart, with no hierarchy between states and no concurrent regions. In most cases this leads to an exponential number of states and transitions in the FSA, but because the process is completely automatic and there is no need to manually inspect the resulting FSA, this aspect is in many cases not relevant. Because MIGtool does not use executable UML, transition guards are not given semantics and are simply treated as part of the event label of a transition. MIGtool produces an XML representation (graphML) of the resulting FSA, the *interaction graph*. It is a directed multigraph[1], potentially with cycles and loops, with edges labeled with the name of the corresponding action. In the following we will be using as synonymous the terms state/vertex, action/edge and trace/path.

3.2 Grounding usage scenarios

Because in all but the most trivial interaction graphs there are loops or cycles, the set of possible interaction traces is infinite. For this reason, scenarios need to be defined and used as constraints on the possible execution traces that can be generated. Users of MIGtool define interaction scenarios by specifying key steps (called bridge sets) that users are expected to go through; we call this process *grounding usage scenarios on models*. A *bridge set* is a subset of the edges of the interaction graph; a well formed bridge set is a non-empty set (an empty bridge set would make the scenario unviable). In general a specification of a scenario includes an initial state and a non-empty sequence of bridge sets. For example, to specify a scenario for replying to an email message, assuming an initial state that corresponds to a user interface that displays all messages in the inbox, one could select all the edges associated to the action `reply` (bridge set 1), followed by edges labeled with `typeBody` (bridge set 2), followed by edges labeled with `send` (bridge set 3). In this way scenarios with cycles can be formulated (*e.g.*, reply twice to two messages). A *stage* of a scenario comprises two consecutive bridge sets. To cope with multi edges, the interaction graph is *simplified*: all edges between a pair of vertices are merged into a single one, whose label includes the original ones.

3.3 Searching traces

Quite expressive languages can be conceived for grounding scenarios (*e.g.*, regular expressions on sequences of action labels). But such expressivity bonus needs to be balanced with computational tractability: even models with a dozen states might correspond to interaction graphs with several hundred states and several thousand edges, leading to an enormous number of possible traces to filter even for scenarios with just a few stages.

To cope with this we implemented a trace searching algorithm that processes each of the stages sequentially, starting from the initial state. Given a stage i and a bridge set B_i, the algorithm does a breadth-first search of all the geodesic paths that connect *each* of the ending vertices of edges in B_i with *some* of the starting vertices of edges in B_{i+1}. If some bridge in B_{i+1} cannot be reached, then it is dropped from further searches. MIGtool creates a new graph from the geodesic paths found for each stage, and then joins these graphs so that geodesic paths found for stage i are joined with those of $i + 1$. These global paths, connecting the initial state to reachable bridges of the last bridge set of the scenario, are called *execution traces*.

Notice that a scenario specifies only the desired occurrences of actions, not all the necessary ones. For example, if the model prescribes that in order to perform action `view(message)` while reading another message one has to `goBack` to the inbox first, a scenario specifying two consecutive `view(message)` would lead to traces that include also the `goBack` action, even if that step is not explicitly specified in the scenario. It is the task of MIGtool to unfold paths in the graph and search all the geodesic paths that connect the desired user actions.

3.4 Detour traces

The algorithm described so far finds (some of) the global paths connecting the initial state to one or more bridges for each of the specified stages. The globally shortest paths are included in the solution, together with other viable alternatives. These traces are called *detour order 0* traces.

MIGtool is used to process a model and a scenario and to generate execution traces of detour order = 0, ..., H. For each stage, up to a maximum detour order H, the search algorithm creates traces with detour order $k + 1$ by collecting the set D_k of states with order 0 up to k, and by identifying the neighbors N_k of D_k (N_k is the set of states not included in D_k that can be reached through an edge from some state in D_k). Edges connecting D_k with N_k represent deviations that users might follow, and geodesic paths from N_k to D_k constitute recovery paths that users might follow to complete the scenario from states in N_k. These deviations and recovery paths are joined and constitute the traces of order $k + 1$. It could happen that for some state in N_k there is no path leading to any vertex in D_k: in such a case the deviation is a dead end that prevents the user to complete the scenario.

The time complexity of the search algorithm is $O(KM(E + V)))$, where there are K bridge sets, their mean size is M, the interaction graph has V vertices and E edges. Therefore it scales well with the size of the interaction graph and/or complexity of the scenarios. In practice, for interaction graphs consisting of about 10000 edges and a dozen of bridge sets, on a low-cost PC it takes about 20 seconds to generate traces of order 0 to 3.

4. COMPARING APPLICATIONS

In this section we describe some examples of the results that can be obtained with MIGtool, based on well known web mail clients, namely Gmail, Horde, SquirrelMail and Roundcube. We chose these examples because they are well known, and therefore are easy to describe and understand. And yet, despite email being a very well understood domain, the kind of questions that can be posed and the answers that are found provide interesting insights on some of the usability properties of these applications.

[1]A directed multigraph is a directed graph such that there are 2 or more edges that have the same end points. Cycles are paths that include 2 or more occurrences of the same vertex. Loops are edges that start and end on the same vertex. "Geodesic path" is a synonymous term with "shortest path".

4.1 Models and scenarios

Models of the four applications have been manually defined. In order to support a fair comparison, all four models cover the same set of use cases at the same level of detail: listing the content of the inbox, reading a message or conversation, replying to a message, composing and sending a new message.

Figure 1 shows part of the model of Gmail. At some point, a user is viewing all the conversations of the inbox (state `viewingConversations`); available actions include moving to the next or previous block of conversations, refreshing the list, or opening a specific conversation (transition `open(conversation)`). This transition is assumed to occur when the user clicks on any one of the visible conversations - it is as if there is a collection of transitions, each one for a different conversation that can be opened; the parameter `conversation` is conceptually bound to any occurrence of a conversation. Such transitions lead to the state `viewingAConversation`, where the behavior of the system is defined by a more detailed state machine. By default the user is viewing a conversation, but by performing the `reply` action the UI moves to a state called `replying`, where the body of the reply can be typed, the subject can be changed, or another addressee can be added.

This behavior "happens" in one of the two concurrent regions specified by this model (left and right side of Figure 1). In parallel to this, the user can either be reading messages (state `reading`) or may be composing a new message (state `composing`). In the latter case the user can independently add recipients, attachments, write the body or subject of the message; and send or cancel it.

The model that we show here is part of what we used in the examples reported below. The actual model that we used consists of 24 states, 12 pseudo states, 12 regions, and 61 transitions. This model is similar to that being discussed in IFML examples mentioned by [26].

The flattening process, which takes a couple seconds on a low-cost PC, produces a directed multigraph with 47 vertices and 634 edges; when simplified by collapsing multiple edges, the number of edges drops to 312. Each vertex represents one of the possible combinations of simple states in any of the regions that can be active at the same time.

Similar models and corresponding graphs were produced for the other three applications.

4.2 Analysis of interaction designs

Inspection of the interaction graph is not particularly useful because even for small graphs like the one obtained from our Gmail model no particular structure is evident that was not known from the model. Because of the large number of cycles that exist among states, edges in the interaction graph form an intricate web of possible action sequences. We believe this greatly reduces usefulness of typical network analysis metrics, such as *betweenness, eccentricity, page-rank* and *eigenvalue* centrality measures.

To be able to obtain results that bear upon usability, we process further the graph, by specifying scenarios and computing traces. The scenario, which assumes `ViewingConversations` is the initial state, is specified with the following 11 bridge sets:

```
1  open(conversation)
2  open(conversation)
3  reply
4  typeBody
5  send
6  compose
7  addRecipient
8  open(files)
9  writeBody
10 writeSubject
11 send
```

In plain language such a scenario means opening a conversation, doing something and then opening a second one, replying to the last message of the second conversation by typing the body of the response, sending it, and then composing a new message by adding a recipient, an attachment, typing the body and subject, and finally sending it. Notice that because the model precludes the possibility that there are two *consecutive* actions called `open(conversation)`, MIGTool has to find traces such that between bridge set 1 and 2 there is at least one `goBack` action.

The execution traces for such a Gmail scenario, with a detour limit of 3, consist of a graph with 474 vertices and 1753 edges. These traces entail 12 geodesic paths with order 0, 82 with order 1, 30 with order 2 and 33 with order 3. Figure 2 shows the number of paths obtained for the four applications, split by detour order.

One of the order 0 geodesic paths is shown below; each line represents an action that the user is expected to perform, which is entailed by the model. Actions marked with "*" are those not specified in the bridge sets and found in the interaction graph.

```
1  open(conversation)
2  goBack *
3  open(conversation)
4  reply
5  typeBody
6  send
7  compose
8  attachFiles *
9  addRecipient[first]
10 open(files)
11 writeBody
12 writeSubject
13 send
```

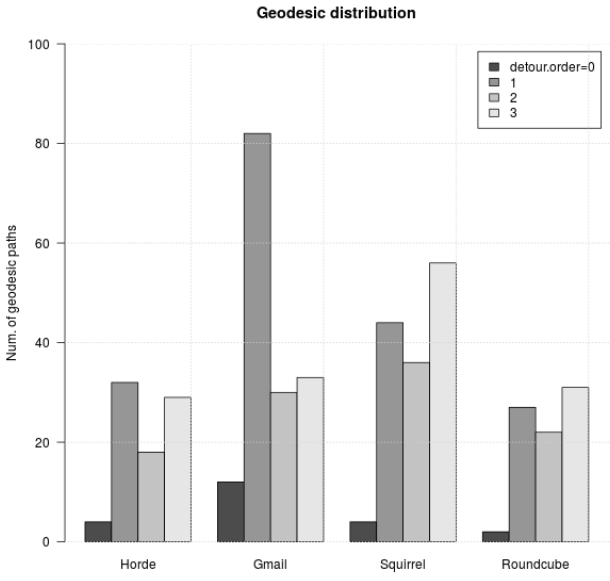

Figure 2: Number of geodesic paths split by detour order.

Compared to the other applications, Gmail has the highest num-

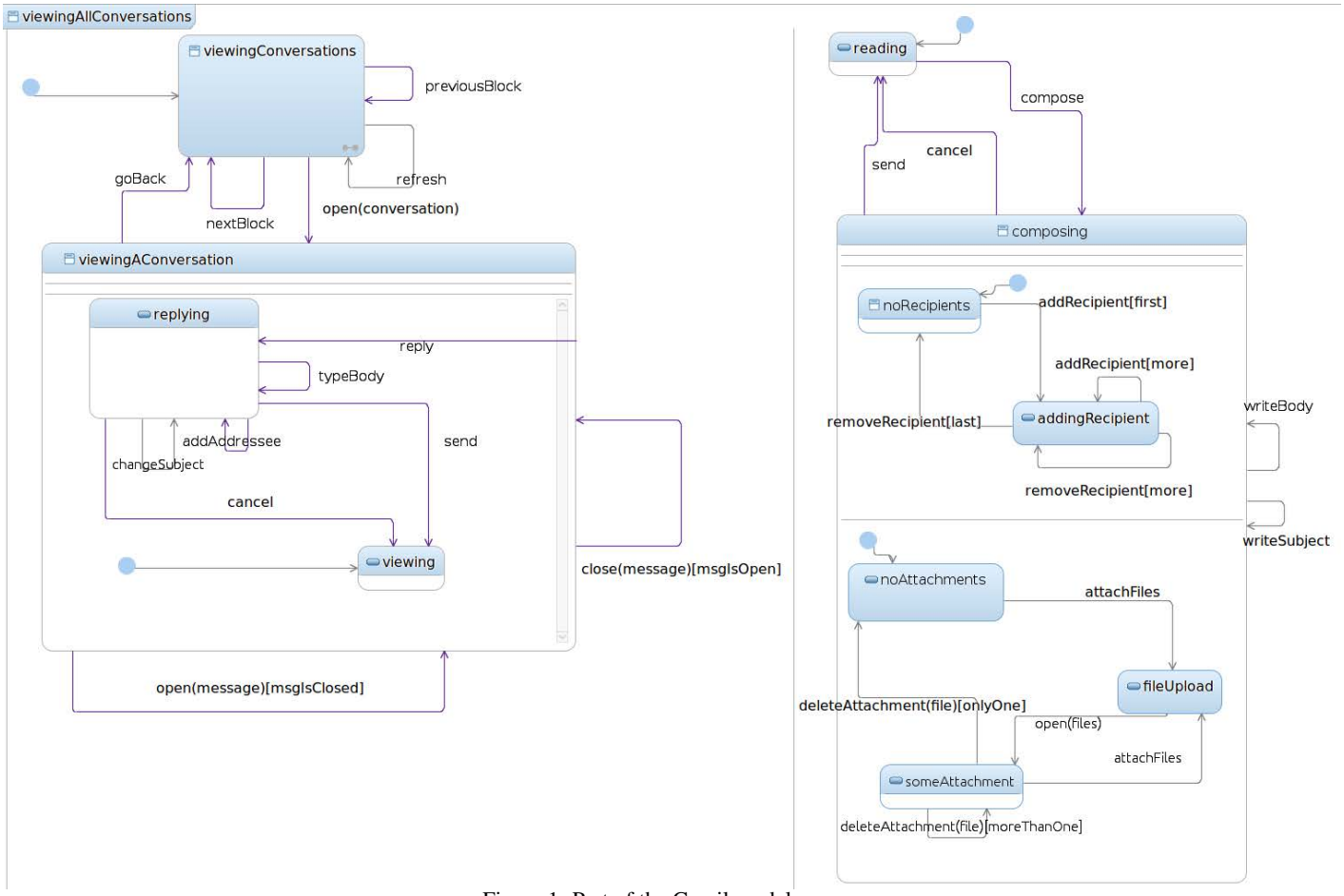

Figure 1: Part of the Gmail model.

ber of order 0 and 1 traces; it has the highest difference between the number of order 0 and 1, and between order 1 and 2 traces (at least 3 times as many order 0 paths than any of the other applications, and at least twice as many order 1 paths as the any of the other applications). In other words, Gmail offers 3 times as many error-free alternative paths to accomplish the scenario, which indicates that users might more easily follow one of those paths, than when using other applications. Notice that Roundcube has the smallest number of order 0 paths (2 of them), which means that users are not given much flexibility and freedom in carrying out correctly the scenario. However, Gmail provides also almost twice as many order 1 paths, which means that users could be more easily induced into an erroneous path than when using another system (or, as an alternative interpretation, users could be given more flexibility). Because the number of order 2 or 3 paths decreases, Gmail reduces therefore the "error proneness" of this UI, for executions that involve 2 or 3 errors.

For none of the systems a detour leads to dead-ends.

Figure 3 shows the length of paths in the best case, i.e., when users would always choose the shortest route. Gmail offers the shortest paths across the four detour orders (for order 0 the length is 13 steps, saving more than 13% steps compared to other applications; for order 3 the length is 17 steps; the other applications are remarkably similar among them). A plausible interpretation is therefore that Gmail not only offers many more error-free paths, but also gives the shortest ones. Users are given more flexibility and more efficiency. Because also paths with order 1 or more are the shortest ones among the four applications, Gmail also makes users more efficient in recovering from errors.

To combine these two results, we can easily compute the frequency of paths having different length within an application. Figure 4 shows, for each application, the frequency of order 0 paths, the frequency of paths with length less or equal to 15 (the minimum length across the four applications), and the frequency of optimal paths (the shortest ones). These values show that Gmail users have the lowest probability to hit an order 0 path (because of the relative large number of order 1 paths made available by Gmail), have the lowest probability of hitting the shortest paths (10 times smaller than the best of the other applications), but have the highest probability to hit a path with length 15 or less. Thus, the flexibility and efficiency that can be exploited with Gmail are counterbalanced by the required knowledge and capability of choosing an optimal path. In particular, Gmail offers many detours of order 1 which increase flexibility for some users and might decrease effectiveness for less skilled ones.

Another probabilistic analysis can be performed using page rank, which computes the probability that a random walk in a graph visits a certain vertex [29]. We computed the page rank (with a damping value of 5% - meaning that the random walker with probability 5% jumps to an arbitrary state and probability 95% chooses one of the actions available in the current state) for each vertex in the graph, and then summed up page rank values for vertices with different detour order. Figure 5 shows the resulting sums.

With Gmail the probability of visiting an order 0 state is close to

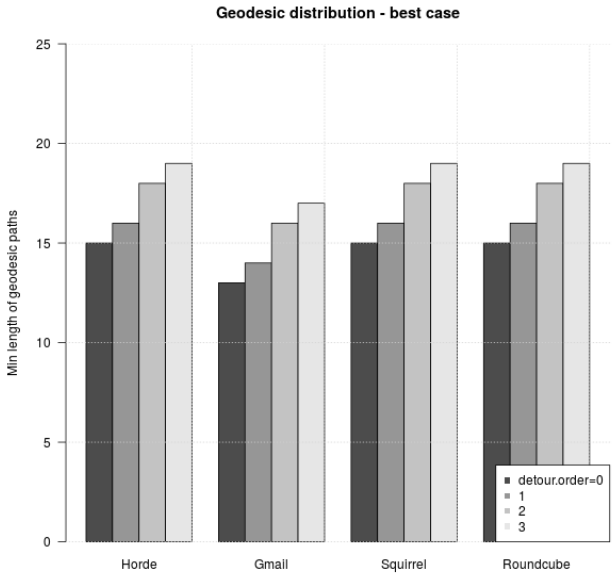

Figure 3: Minimum length of geodesic paths split by detour order.

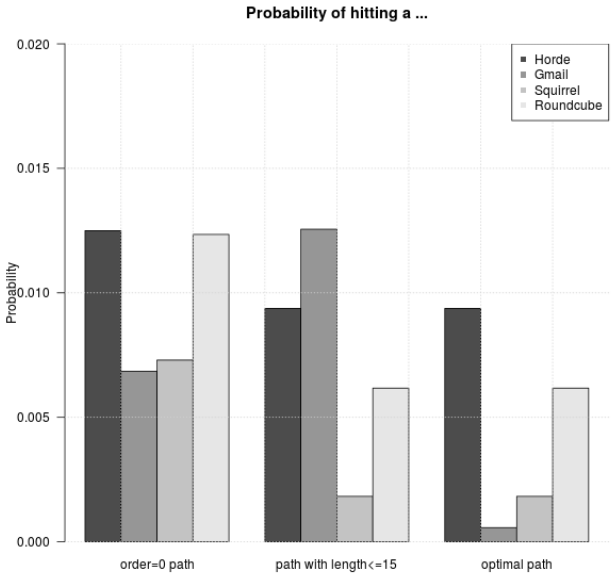

Figure 4: Frequency of an optimal path.

70%, the lowest among the four applications. But when it comes to visiting an order 0 or 1 state, the probability increases to 87%, which is the highest. Thus, in a comparison of Gmail against SquirrelMail, a completely random usage of SquirrelMail has 10% more probability of hitting an error-free state than Gmail. That advantage is slightly reduced when considering order 0 and 1 paths, because with SquirrelMail the probability is 85% and Gmail it is 87%. This means that somebody with no knowledge on how to use an email front end, when using Gmail would have 10% fewer chances of carrying out the scenario without making any error, as opposed to when using SquirrelMail: SquirrelMail provides more guidance. Across the four UIs, the probability of making at most 1 error is approximately the same[2].

Manual inspection of the shortest paths indicates that one reason for the greater potential efficiency offered by Gmail is due to the fact that users can start composing a new message while reading a conversation, whereas in other applications an explicit "close action" of the reading activity has to be performed. Another reason relies on the more streamlined process to attach a file: in Gmail one needs to select the file(s) and they are automatically uploaded, whereas in other applications one has to explicitly perform the uploading step after selecting them.

Figure 6 shows the *action density* of the four applications, *i.e.* the average number of actions per state involved in traces (defined by the out-degree), and average number of *unique* actions per state. The former is an overall measure of the number of actions that are made available by a UI, the latter can be used to analyze how many *new* options the user is presented with in any state. For our examples, the values are all in the range between 5.9 actions/state in the case of Gmail and 7.1 for Horde, and 1.3 unique actions for SquirrelMail and 2 for Horde. This suggests that Gmail features a more compact design (fewer actions to do the same things), and SquirrelMail is even more compact when it comes to the different types of actions; thus it could be easier to learn.

[2]These results are not sensitive to the value chosen for the damping factor: differences across the four applications are stable when d ranges in $[0.05, 0.20]$.

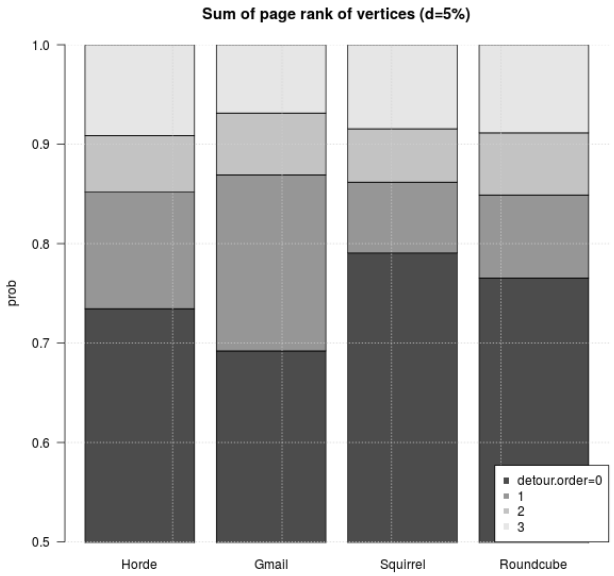

Figure 5: Probability that a random walk visits detour 0, 1, 2 or 3 states.

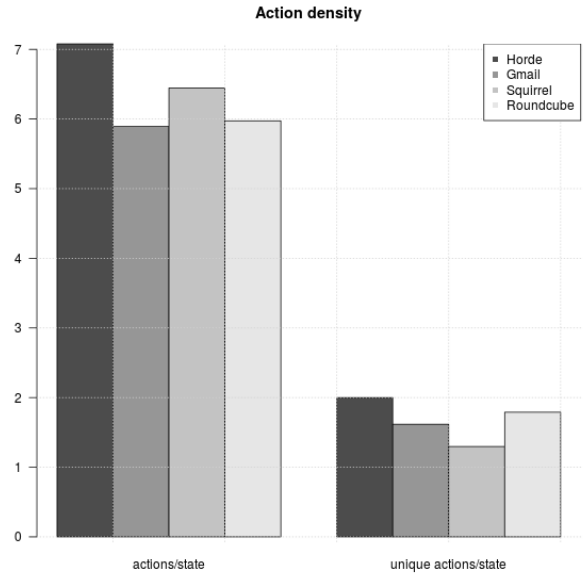

Figure 6: Action density.

4.3 Comparing scenarios

Previous examples show how MIGtool can be used to compare different UIs against the same scenario. The same kind of analysis can be carried out to assess how suitable a design is for different scenarios. For example, a designer might be interested in a comparison of replying to a message (scenario "reply") as opposed to composing a new one ("compose"), given a single UI.

Let's focus now on Horde. It turns out that "compose" entails many more order 0 paths than "reply" (154 vs 86), and a comparable number of higher order paths. The path length in the best case is the same across the two scenarios, but in the average case "compose" has a length of 7.75 steps compared to 8.5 for "reply". Probability of hitting an optimal path is 4 times higher for "compose".

This means that users will have twice as many choices between correct paths when composing a new message rather than simply replying to a read one. On average, when composing, users could be 9% more efficient, and they are 4 times more likely to do the right things. Thus, Horde is more suitable for composing new messages than it is for replying to existing ones.

4.4 Adding features

Execution traces can be used also to assess what is the effect of adding a widget or feature to an existing UI. For example, we studied cruise control features of cars. One of the examples is system S[3], where the driver can engage the system, and once it is engaged, set speed can be increased or decreased with small or large steps. Of course the system can also be disengaged (by pressing the brake pedal or by acting on a lever). System A is more elaborate, as it includes also a memory function: when it is disengaged it remembers the currently set speed, which can be recalled later on. There are two ways to re-engage it: one by setting a new speed, and one by recalling the previous one. In addition, if the car drives for more than 5 minutes at a higher speed than the set one, system A automatically disengages.

Thus, one possible design question is "What are the effects of adding these functionalities?" in a typical driving scenario where a speed is selected, then the system is disengaged, and later on the same speed needs to be set.

[3]There is no need to disclose the actual brand.

System	N_0	L_0	N_1	L_1	p	AD	UAD
S	1	7	8	9.4	11%	1.75	1.25
A	2	4.5	7	6.8	6%	2.20	1.20

Table 1: Comparison of the two cruise control designs. N_i: number of execution traces with detour order i; L_i: average length of traces with detour order i; p: probability to hit the shortest execution trace; AD: actions/state; UAD: unique actions/state.

When using a scenario for S where we assume that re-setting the speed is done manually by the driver with 4 actions on "up" and "down" (to approximately set the previous speed), the comparison produces the values shown in Table 1. System A has 2 alternative optimal paths, their length is 4.5 (36% shorter than in S); both systems have a comparable number of detour 1 traces (N_1), but their length in A (L_1) is 28% shorter. The probability of hitting the optimal trace is however twice as large in S (p=11%), which features also a more compact design (AD=1.75). In both cases detours are caused by the possibility of disengaging the system at the wrong moment.

Therefore we can conclude that: (1) system A makes users more efficient (a saving of 36%) for the considered scenario; (2) with A there are two possible ways to achieve the scenario, thus more flexibility is given; (3) with A the probability of doing the right thing is almost half of system S: it might be more difficult to do the right thing because more possibilities are offered; (4) system S features a more compact design, with fewer actions to be performed at each moment.

5. DISCUSSION

An important issue underlying MIGtool is the modeling effort that is needed upfront. Our experience, based on several case studies and some industrial examples, is that models do not need to be complete representations of the behavior of the application under study. By following an agile modeling approach [1], models can be easily developed by one person in less than one day, using any UML capable design tool. Even more complex models (in our experience up to 200 states and 450 transitions) can be developed and verified in 3 days by one individual. Experience in using statecharts to model behavior of UIs is needed though; useful suggestions are given in [18, 38].

UML state diagrams provide a very expressive language, well suited to specify behavior of UIs based on discrete events. in MIGtool all the UML language [27] for state machine has been used, including sundry pseudostates. Even though there are fundamental limits (inability to handle undo/redo's, because this goes beyond a finite state representation; inability to handle customizable toolbars, because this requires models that change at runtime; inability to handle perceptive UIs, because they are not well suited to be described in terms of discrete states), in many practical cases they can be be isolated and/or ignored [6]. One particular limitation of MIGtool stems from the fact that it does not rely on an executable UML metamodel; as a consequence behaviors that rely on the fact that firing a transition broadcasts new events cannot be properly represented and analyzed. For the same reason the meaning of guards and actions is not considered by MIGtool, and they are treated simply as parts of the labels.

Expressivity of the modeling language means that different designers are likely to produce different models for the same UI. As a consequence, it is possible that metrics computed by MIGtool

do depend on different modeling choices. It is worth mentioning, though, that because of the flattening process, several differences are reduced (for example, those dealing with using a different hierarchy of states, or with differently distributed concurrent regions across states), and sensitivity is correspondingly reduced. Future studies for addressing this issue could follow what was done in [19].

Differently from other model-based approaches, such as those based on IFML, MIGtool uses *only* a model of the behavior of the UI. In IFML the modeler specifies behavior in terms of concepts that are very close to statecharts (IFML adopts a dual representation of AND/OR-states, has guards, has events, has an underlying event broadcasting mechanism) upon which components with their data binding are defined, so that transitions can be easily bound to data instances. On the other hand, designers using MIGtool do not need to cope with data modeling, nor with decisions dealing with presentation. In a sense, MIGtool uses only the *controller* part of the Model-View-Controller paradigm that is adopted when developing UIs. Although in principle the same results could be obtained from IFML models, to do so one needs to isolate the control part of the IFML model from the data part; this is difficult to do because, by design, IFML integrates very well the two aspects.

A designer using MIGtool is free from other concerns that in the end affect usability, and the conclusions that are derived with MIGtool can be combined with other results *after* the analysis is performed. As mentioned above, ours is an agile modeling approach [1] that: (a) enables analysis of a UI well before it is developed; (b) requires *only* a model of its behavior; (c) supports exploration of different tasks/scenarios; (d) enables grounding of tasks/scenarios through a platform-independent specification. As such, grounding can be viewed as the specification of a *reification* relation between a *task-and-domain-concepts* level to an *abstract-user-interface* level (we are using terms of the CAMELEON reference model [9]). It is possible to extend MIGtool so that also task models (specified, for example, through CTT, UsiXML, HAMSTERS [25, 21, 22]) can be processed and used as specification of the scenarios to be analyzed. For example, a CTT model could be annotated by associating subtasks to bridge sets and then used to generate a sequence of bridge sets, from which MIGtool could generate traces and metrics. At that point, MIGtool could be used as a tool to explore suitability of tasks with respect to a behavioral model of a user interface. This problem has been addressed in [22] through a mechanism that relates task models to source code annotations of the implementation of the user interface. In this way it is possible to simulate execution of the tasks in parallel with execution of the system and ensure compatibility of the system with respect to tasks.

In terms of validity of conclusions obtained through MIGtool, because they are devoid of user behavior assumptions (such as preferences, skills, interpretations, ergonomic constraints) they are very general and conservative. On the other hand, they are also generic because they do not consider data and presentation aspects. For example, it is unfeasible to use MIGtool to predict the time needed by a user to complete a scenario. However, as mentioned before, MIGtool can be used to analyze the whole interaction design and gather data to inform more specific analyses that could be performed, for example, with CogTool or CogTool-Helper [37].

At the moment MIGtool does not use weighted edges in the interaction graph. It is easy to extend it to search paths that minimize the total weight, and therefore in such a way to perform analyses that are similar to the ones based on Markov chains suggested in [39].

Likewise it is possible to extend MIGtool to compute also the lostness metric [36], based on states with different detour orders. This would give yet another objective metric to compare different designs. We opted for non doing it in light of the fact that lostness was designed for exploratory activities in information websites, not for goal oriented scenarios like the ones we considered in previous examples.

MIGtool is implemented partly in Java (model processing) and partly in R/iGraph [33]. Currently MIGtool reads UML models represented as XMI files; it could be easily extended to handle also SCXML representations of statecharts [4].

6. CONCLUSION

We presented a method that can be used to support analysis of user interfaces even before they are built or prototyped. UML statecharts representing the intended behavior of user interfaces are processed and matched to interaction scenarios, producing a number of graph-theoretic metrics defined on possible execution traces.

We showed that with this approach one can compare different designs against the same scenario, or the same design against different scenarios. In both situations precise, quantitative and objective measures can be generated regarding flexibility offered to users, their potential efficiency, and error proneness of the user interface.

The method should not be used to draw final usability conclusions, as it is devoid of any concerns dealing with what is presented to users, how they could perceive, understand, and manipulate that. But, because the approach can be applied when just a specification of the user interface is available, it supports construction and engineering of interactive systems and could help in iteratively improving a design before building any UI prototype.

7. REFERENCES

[1] S. Ambler. 2002. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. Wiley.

[2] C. Appert and M. Beaudouin-Lafon. 2008. SwingStates: adding state machines to Java and the Swing toolkit. *Software: Practice and Experience* 38, 11 (2008), 1149–1182.

[3] E. Barboni, S. Conversy, D. Navarre, and P. Palanque. 2007. Model-based engineering of widgets, user applications and servers compliant with ARINC 661 specification. In *Interactive Systems. Design, Specification, and Verification*. Springer, 25–38.

[4] J. Barnett and et al. 2015. *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. Technical Report. W3C. http://www.w3.org/TR/2015/REC-scxml-20150901.

[5] R. Bellamy, B. John, and S. Kogan. 2011. Deploying CogTool: integrating quantitative usability assessment into real-world software development. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 691–700.

[6] G. Brajnik and S. Harper. 2015. Detaching control from data models in model-based generation of user interfaces. In *Proc. of Int. Conf. on Web Engineering*, F. Frasincar (Ed.). IEEE, Rotterdam, Netherlands.

[7] L. C Briand, Y. Labiche, and J. Cui. 2005. Automated support for deriving test requirements from UML statecharts. *Software & Systems Modeling* 4, 4 (2005), 399–423.

[8] B. Buxton. 2007. *User Experience: Getting the Design Right and the Right Design*. Morgan Kaufmann.

[9] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. 2003. A Unifying Reference Framework for Multi-Target User Interfaces. *Interacting with Computers* 15, 3 (2003), 289–308.

[10] S. Ceri, P. Fraternali, and A. Bongio. 2000. Web Modeling Language (WebML): a modeling language for designing web sites. *Computer Networks* 33 (2000), 137–157.

[11] L.L. Constantine and L.A.D. Lockwood. 1999. *Software for use: a practical guide to the models and methods of usage-centered design*. Addison-Wesley.

[12] C. Fayollas, C. Martinie, P. Palanque, Y. Deleris, J-C. Fabre, and D. Navarre. 2014. An approach for assessing the impact of dependability on usability: application to interactive cockpits. In *Tenth European Dependable Computing Conference (EDCC), 2014*. IEEE, 198–209.

[13] W.-T. Fu and P. Pirolli. 2007. SNIF-ACT: A cognitive model of user navigation on the World Wide Web. *Human–Computer Interaction* 22, 4 (2007), 355–412.

[14] J. Gómez, C. Cachero, and O. Pastor. 2001. Conceptual Modeling of Device-Independent Web Applications. *IEEE MultiMedia* 8, 2 (2001), 26–39. DOI: http://dx.doi.org/10.1109/93.917969

[15] M. Heymann and A. Degani. 2007. Formal analysis and automatic generation of user interfaces: approach, methodology, and an algorithm. *Human Factors: The Journal of the Human Factors and Ergonomics Society* 49, 2 (2007), 311–330.

[16] D. M Hilbert and D.F. Redmiles. 2000. Extracting usability information from user interface events. *ACM Computing Surveys (CSUR)* 32, 4 (2000), 384–421.

[17] K. Hornbæk. 2006. Current practice in measuring usability: Challenges to usability studies and research. *International Journal of Human-Computer Studies* 64, 2 (2006), 79–102.

[18] I. Horrocks. 1999. *Constructing the User Interface with Statecharts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[19] B.E. John. 2010. Reducing the Variability between Novice Modelers: Results of a Tool for Human Performance Modeling Produced through Human-Centered Design. In *Proc. of the 19th Conf. on Behavior Representation in Modeling and Simulation*. Charleston, SC, USA, 95–102.

[20] N. Koch and A. Kraus. 2002. The expressive power of UML-based web engineering. In *Second International Workshop on Web-Oriented Software Technology (IWWOST02)*, D. Schwabe, O. Pastor, G. Rossi, and L. Olsina (Eds.). 105–199.

[21] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López Jaquero. 2004. UsiXML: a Language Supporting Multi-Path Development of User Interfaces. In *Proc. of 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems, EHCI-DSVIS'2004 (Lecture Notes in Computer Science)*, Vol. 3425. Springer-Verlag, Hamburg, Germany, 200–220.

[22] C. Martinie, D. Navarre, P. Palanque, and C. Fayollas. 2015. A generic tool–supported framework for coupling task models and interactive applications. In *Proc. of Engineering Interactive Computing Systems, EICS '15*. ACM, ACM Press, Duisburg, Germany, 244–253.

[23] M. McCurdy, C. Connors, G. Pyrzak, B. Kanefsky, and A. Vera. 2006. Breaking the fidelity barrier: an examination of our current characterization of prototypes and an example of a mixed-fidelity success. In *CHI 2006*. ACM, ACM Press, New York, NY, 1233–1242.

[24] S. Meliá, J. Gómez, S. Pérez, and O. Díaz. 2008. A Model-Driven Development for GWT- Based Rich Internet Applications with OOH4RIA. In *Proc. 8th Int'l Conf. Web Eng. (ICWE 2008)*. IEEE CS Press, 13–23.

[25] G. Mori, F. Paternò, and C. Santoro. 2002. CTTE: Support for Developing and Analysing Task Models for Interactive System Design. *IEEE Transactions on Software Engineering* 28, 8 (August 2002), 797–813.

[26] OMG. 2013. *Interaction Flow Modeling Language (IFML), FTF – Beta 1* (omg document number: ptc/2013-03-08 ed.). Technical Report. OMG. http://www.omg.org/spec/IFML/1.0

[27] Object Management Group OMG. 2015. OMG Unified Modeling Language (OMG UML) Version 2.5. http://www.omg.org/spec/UML/2.5. (March 2015). http://www.omg.org/spec/UML/2.5/PDF

[28] M. Otter and H. Johnson. 2000. Lost in hyperspace: metrics and mental models. *Interacting with Computers* 13, 1 (2000), 1–40.

[29] L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. The PageRank citation ranking: bringing order to the Web. (1999).

[30] D.L. Parnas. 1969. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *ACM '69 Proc. of the 1969 24th National Conference*. ACM.

[31] P. Pinhero da Silva and N.W. Paton. 2003. User Interface Modeling in UMLi. *IEEE Software* (2003), 62–69.

[32] J. Preece, Y. Rogers, and H. Sharp. 2002. *Interaction Design*. John Wiley and Sons.

[33] R Core Team. 2014. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/

[34] J. Rubin and D. Chisnell. 2008. *Handbook of Usability Testing* (second ed.). Wiley.

[35] J.L. Silva, C. Fayollas, A. Hamon, P. Palanque, C. Martinie, and E. Barboni. 2014. Analysis of WIMP and Post WIMP Interactive Systems based on Formal Specification. *Electronic Communications of the EASST* 69 (2014).

[36] P.A. Smith. 1996. Towards a practical measure of hypertext usability. *Interacting with Computers* 8, 4 (1996), 365–381.

[37] A. Swearngin, M.B. Cohen, B.E. John, and K.E. Bellamy. 2013. Human Performance Regression Testing. In *Proc. of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 152–161. http://dl.acm.org/citation.cfm?id=2486788.2486809

[38] H. Thimbleby. 2007. *Press on: principles of interaction programming*. The MIT Press.

[39] H. Thimbleby, P. Cairns, and M. Jones. 2001. Usability analysis with Markov models. *ACM Transactions on Computer-Human Interaction (TOCHI)* 8, 2 (2001), 99–132.

[40] H. Thimbleby and P. Oladimeji. 2009. Social Network Analysis and Interactive Device Design Analysis. In *Proc. of Engineering Interactive Computing Systems 2009*. ACM Press, 91–100.

[41] M. Winckler and P.A. Palanque. 2003. StateWebCharts: A Formal Description Technique Dedicated to Navigation Modelling of Web Applications.. In *Design, Specification, and Verification: 10th International Workshop, DSV-IS'03*. Funchal,Madeira Island, Portugal, 61–76.